High Performance Linear Algebra

Lecture 13: LAPACK and ScaLAPACK numerical algorithms

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

Monday 02, 2026 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

A recap of the previous lectures and today's plan

1 Last time on High Performance Linear Algebra

We have

- Reviewed the basic concepts of shared memory programming with OpenMP,
- Reviewed the basic concepts of distributed memory programming with MPI,
- We have discussed the BLAS for dense matrices and vectors:
  - All three levels of the BLAS in shared memory,
  - All three levels of the BLAS in distributed memory.

The plan for the next two lectures is to:

- Look at the LAPACK library for dense linear algebra in shared memory,
- Look at the ScaLAPACK library for dense linear algebra in distributed memory.

Table of Contents

2 LAPACK: Linear Algebra PACKage

LAPACK: Linear Algebra PACKage

- LAPACK is a software library for numerical linear algebra that provides routines for:
  - solving systems of linear equations,
  - least squares problems,
  - eigenvalue problems,
  - singular value decomposition,
  - and other related problems.
- It is designed to be efficient on modern computer architectures, taking advantage of cache memory and vectorization.
- LAPACK is written in Fortran and is widely used in scientific computing applications.
- It is built on top of the BLAS (Basic Linear Algebra Subprograms) library, which provides low-level routines for performing basic linear algebra operations.

LAPACK: solving systems of linear equations

Two types of driver routines are provided for solving systems of linear equations:

- a simple driver (name ending `-SV`), which solves the system $AX = B$ by factorizing A and overwriting B with the solution X;
- an expert driver (name ending `-SVX`), which can also perform the following functions (some of them optionally):
  - solve $A^\top X = B$ or $A^H X = B$ (unless A is symmetric or Hermitian);
  - estimate the condition number of A, check for near-singularity, and check for pivot growth;
  - refine the solution and compute forward and backward error bounds;
  - equilibrate the system if A is poorly scaled.

The **expert driver** requires roughly twice as much storage as the simple driver in order to perform these extra functions.

LAPACK: solving systems of linear equations

Two types of driver routines are provided for solving systems of linear equations:

- a simple driver (name ending `-SV`), which solves the system $AX = B$ by factorizing A and overwriting B with the solution X;
- an expert driver (name ending `-SVX`), which can also perform the following functions (some of them optionally):
  - solve $A^\top X = B$ or $A^H X = B$ (unless A is symmetric or Hermitian);
  - estimate the condition number of A, check for near-singularity, and check for pivot growth;
  - refine the solution and compute forward and backward error bounds;
  - equilibrate the system if A is poorly scaled.

Both types of driver routines can handle **multiple right hand sides** (the columns of B).

LAPACK: solving systems of linear equations

Two types of driver routines are provided for solving systems of linear equations:

- a simple driver (name ending −SV), which solves the system $AX = B$ by factorizing A and overwriting B with the solution X;
- an expert driver (name ending −SVX), which can also perform the following functions (some of them optionally):
  - solve $A^\top X = B$ or $A^H X = B$ (unless A is symmetric or Hermitian);
  - estimate the condition number of A, check for near-singularity, and check for pivot growth;
  - refine the solution and compute forward and backward error bounds;
  - equilibrate the system if A is poorly scaled.

Different driver routines are provided to take advantage of special properties or storage schemes of the matrix A, for example, if A is *symmetric*, *triangular*, *banded*, or *tridiagonal*.

LAPACK: the simple drivers ?GESV

For a general (non-symmetric, non-Hermitian) matrix A, the simple driver routine to solve the system $AX = B$ is ?GESV, where the ? is the data-type placeholder.

```
SUBROUTINE ?GESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )
  INTEGER           N, NRHS, LDA, LDB, INFO
  INTEGER           IPIV( * )
  ?                 A( LDA, * ), B( LDB, * )
END SUBROUTINE ?GESV
```

- N is the order of the matrix A,
- NRHS is the number of right hand sides (the number of columns of B),
- A is the coefficient matrix A (on entry) and its LU factorization (on exit),
- IPIV is an integer array of pivot indices,
- B is the right hand side matrix B (on entry) and the solution matrix X (on exit),
- INFO is an integer output variable that indicates success or failure of the routine.

LAPACK: the simple drivers DGESV

Let us look at an example of usage of the simple driver DGESV to **solve** a **system of linear equations**.

A classical source of dense linear systems is the discretization of **integral equations**:

$$\int_a^b K(x,y)u(y)\mathrm{d}y = f(x), \quad x \in [a,b].$$

Using a quadrature rule with nodes y_j and weights w_j, we can approximate the integral as

$$\sum_{j=1}^{n} w_j K(x_i,y_j)u(y_j) \approx f(x_i), \quad i = 1,\ldots,n,$$

which leads to the linear system

$$Au = f, \quad A_{ij} = w_j K(x_i,y_j).$$

Discretization of a Stable Volterra Equation

2 LAPACK: Linear Algebra PACKage

We consider the Volterra integral equation of the **second kind**:

$$u(s) - \int_0^s (s-t)u(t)\mathrm{d}t = \sin(s), \quad s \in [0,1].$$

We discretize the domain into n nodes $s_i = \frac{i-1}{n-1}$ with step size $h = \frac{1}{n-1}$. Using the **composite trapezoidal rule**, the integral is approximated for each node s_i:

$$\int_0^{s_i} (s_i - t)u(t)\mathrm{d}t \approx \sum_{j=1}^{i} w_{ij}(s_i - t_j)u(t_j), \quad t_j = s_j,$$

where the weights are $w_{ij} = \frac{h}{2}$ for the endpoints ($j = 1, i$) and $w_{ij} = h$ otherwise. This leads to a **lower triangular** linear system $(I - A)u = f$:

$$u_i - \sum_{j=1}^{i} A_{ij}u_j = f_i, \quad A_{ij} = w_{ij}(s_i - t_j), \quad i = 1, \ldots, n.$$

LAPACK: writing the function building the matrix A and $\mathbf{b}$

We need to write a function that builds the matrix A and the right-hand side vector b.

```
subroutine build_system( n_size, mat, rhs, xplot_vec, sol_vec )
  use iso_fortran_env, only: dp => real64
  integer, intent(in) :: n_size
  real(dp), intent(out) :: mat(n_size,n_size), rhs(n_size), xplot_vec(n_size),
  ↪  sol_vec(n_size)
end subroutine build_system
```

- `n_size` is the size of the system,
- `mat` is the matrix A,
- `rhs` is the right-hand side vector $\mathbf{b}$,
- `xplot_vec` is the vector of x coordinates for plotting,
- `sol_vec` is the vector of the exact solution for comparison.

LAPACK: writing the function building the matrix A and b

The implementation of the function is as follows:

```
integer :: i, j
real(dp) :: s_i, t_j, h

h = 1.0_dp / real(n_size - 1, dp)
mat = 0.0_dp
!$omp parallel do private(i,j,s_i,t_j) shared(mat,rhs,xplot_vec,sol_vec,h)
do i = 1, n_size
    s_i = real(i-1, dp) * h
    rhs(i) = sin(s_i)  ! f(s)
    xplot_vec(i) = s_i ! x coordinates for plotting
    sol_vec(i) = 0.5*sinh(s_i) + 0.5*sin(s_i)
    mat(i,i) = 1.0_dp  ! Identity part: x(s)
```

LAPACK: writing the function building the matrix A and $\mathbf{b}$


```
    do j = 1, i
      t_j = real(j-1, dp) * h ! Kernel K(s, t) = s - t
      if (j == 1 .or. j == i) then
          mat(i,j) = mat(i,j) - (s_i - t_j) * (h / 2.0_dp)
      else
          mat(i,j) = mat(i,j) - (s_i - t_j) * h
      end if
    end do
end do
!$omp end parallel do
```

- We initialize the matrix and vectors,
- We use OpenMP to parallelize the outer loop over i,
- We compute the entries of the matrix A and the right-hand side vector $\mathbf{b}$.

LAPACK: the solution step

The remaining part of the program uses the LAPACK routine `DGESV` to solve the linear system, after reading the matrix size from the command line and using the `build_system` subroutine to create the matrix and right-hand side vector.

```
program linear_system_solve
   use, intrinsic :: iso_fortran_env, only: wp => real64, error_unit
   implicit none
   character(len=20) :: arg
   integer :: n, info
   real(wp), allocatable :: A(:,:), b(:), x(:), xplot(:), sol(:)
   integer, allocatable :: ipiv(:)

   ! Read matrix size from command line arguments
   if (command_argument_count() < 1) then
```

LAPACK: the solution step


```
    write(error_unit, *) "Usage: linear_system_solve <matrix_size>"
    stop 1
end if
call get_command_argument(1, arg)
read(arg, *) n

! Allocate arrays
allocate(A(n,n), b(n), x(n), xplot(n), sol(n), ipiv(n))

! Initialize matrix A and vector b
call build_system(n, A, b, xplot, sol)
! Solve the linear system A*x = b using LAPACK
x = b
```

LAPACK: the solution step


```
call dgesv(n, 1, A, n, ipiv, x, n, info)

! Output the solution vector x to file "solution.out"
open(unit=10, file="solution.out", status="replace", action="write",
↪  iostat=info)
if (info /= 0) then
   write(error_unit, *) "Error opening output file"
   stop 1
end if
write(10, '(A)') "x computed exact"
do n = 1, size(x)
   write(10, '(E24.16,E24.16,E24.16)') xplot(n), x(n), sol(n)
end do
```

LAPACK: the solution step

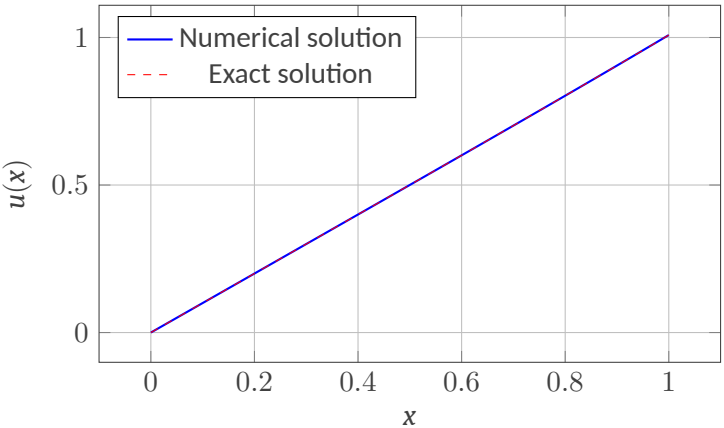

```
close(10)

! Deallocate arrays
deallocate(A, b, x, xplot, sol, ipiv)
stop 0
```

- The program reads the matrix size from command line arguments,
- Allocates the necessary arrays,
- Calls the `build_system` subroutine to initialize the matrix and vector,
- Uses `DGESV` to solve the linear system,
- Outputs the computed solution and exact solution to a file for comparison.

Visualization of the solution of the integral equation

Computing the solution with $n = 100$ nodes and plotting the numerical solution against the exact solution:



A remark on integral equations and dense linear systems

- For large-scale problems, storing and manipulating dense matrices can be memory-intensive and computationally expensive.
- However, the matrices arising from integral equations often have **special structures** that can be exploited:
  - **Low-rank approximations**: The matrix may be well-approximated by a low-rank matrix, reducing storage and computation.
  - **Hierarchical matrices** (H-matrices): Exploit block-wise low-rank structure for efficient storage and computation.
  - **Sparse representations**: Using techniques like the Fast Multipole Method (FMM) to avoid explicit matrix construction.
- These alternative storage formats can significantly reduce memory requirements and computational complexity compared to standard dense matrix representations.

LAPACK: the other linear system routines

Type of matrix and storage scheme	Operation	Single precision real	Single precision complex	Double precision real	Double precision complex
General	simple driver	SGESV	CGESV	DGESV	ZGESV
	expert driver	SGESVX	CGESVX	DGESVX	ZGESVX
General band	simple driver	SGBSV	CGBSV	DGBSV	ZGBSV
	expert driver	SGBSVX	CGBSVX	DGBSVX	ZGBSVX
General tridiagonal	simple driver	SGTSV	CGTSV	DGTSV	ZGTSV
	expert driver	SGTSVX	CGTSVX	DGTSVX	ZGTSVX
Sym./Herm. pos. def.	simple driver	SPOSV	CPOSV	DPOSV	ZPOSV
	expert driver	SPOSVX	CPOSVX	DPOSVX	ZPOSVX

LAPACK: the other linear system routines

Type of matrix and storage scheme	Operation	Single precision real	Single precision complex	Double precision real	Double precision complex
Sym./Herm. pos. def. (packed)	simple driver	SPPSV	CPPSV	DPPSV	ZPPSV
	expert driver	SPPSVX	CPPSVX	DPPSVX	ZPPSVX
Sym./Herm. pos. def. band	simple driver	SPBSV	CPBSV	DPBSV	ZPBSV
	expert driver	SPBSVX	CPBSVX	DPBSVX	ZPBSVX
Sym./Herm. pos. def. tridiagonal	simple driver	SPTSV	CPTSV	DPTSV	ZPTSV
	expert driver	SPTSVX	CPTSVX	DPTSVX	ZPTSVX
Sym./Herm. indefinite	simple driver	SSYSV	CHESV	DSYSV	ZHESV
	expert driver	SSYSVX	CHESVX	DSYSVX	ZHESVX

LAPACK: the other linear system routines

Type of matrix and storage scheme	Operation	Single precision real	Single precision complex	Double precision real	Double precision complex
Complex symmetric	simple driver		CSYSV		ZSYSV
	expert driver		CSYSVX		ZSYSVX
Sym./Herm. indefinite (packed)	simple driver	SSPSV	CHPSV	DSPSV	ZHPSV
	expert driver	SSPSVX	CHPSVX	DSPSVX	ZHPSVX
Complex symmetric (packed)	simple driver		CSPSV		ZSPSV
	expert driver		CSPSVX		ZSPSVX

- The table summarizes the LAPACK routines for solving systems of linear equations for various types of matrices and **storage schemes**.

LAPACK: storage schemes

Generally, LAPACK supports different storage schemes for matrices to optimize memory usage and computational efficiency. The main storage schemes are:

- **Full storage**: The entire matrix is stored in a two-dimensional array. This is the most straightforward representation but can be inefficient for large matrices.
- **Banded storage**: Only the non-zero bands of a banded matrix are stored,
- **Packed storage**: Only the non-zero elements of symmetric or Hermitian matrices are stored in a one-dimensional array,

We have already seen examples of **full storage**, let us briefly discuss the other storage schemes.

LAPACK: banded storage scheme

In the **banded storage scheme**, only the non-zero bands of a banded matrix are stored in a compact form.
A banded matrix has non-zero elements only within a certain bandwidth around the main diagonal.

- `KL` is the number of subdiagonals (non-zero elements below the main diagonal),
- `KU` is the number of superdiagonals (non-zero elements above the main diagonal),
- `LDAB` is the leading dimension of the array `AB`, which must be at least `2*KL+KU+1`,
- `AB(KL+KU+1+i-j,j) = A(i,j)` for `max(1,j-KU)<=i<=min(N,j+KL)`

For example, a matrix with $KL = 1$ (one subdiagonal) and $KU = 2$ (two superdiagonals) would be stored in a compact form requiring only $2 \times 1 + 2 + 1 = 5$ rows instead of storing the full matrix.

LAPACK: banded storage scheme example.

Consider the following banded matrix A with $KL = 1$ and $KU = 2$:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

In **banded storage**, this matrix would be stored in the array `AB` as follows:

$$AB = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} \\ a_{12} & a_{23} & a_{34} & a_{45} & 0 \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & 0 \end{bmatrix}$$

Here,

- the first row contains the second superdiagonal,
- the second row contains the first superdiagonal,
- the third row contains the main diagonal,
- the fourth row contains the first subdiagonal.

LAPACK: banded storage scheme example.

Consider the following banded matrix A with $KL = 1$ and $KU = 2$:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

In **banded storage**, this matrix would be stored in the array `AB` as follows:

$$AB = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} \\ a_{12} & a_{23} & a_{34} & a_{45} & 0 \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & 0 \end{bmatrix}$$

Here,

- the first row contains the second superdiagonal,
- the second row contains the first superdiagonal,
- the third row contains the main diagonal,
- the fourth row contains the first subdiagonal.

LAPACK: banded storage scheme example.

Consider the following banded matrix A with $KL = 1$ and $KU = 2$:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

In **banded storage**, this matrix would be stored in the array `AB` as follows:

$$AB = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} \\ a_{12} & a_{23} & a_{34} & a_{45} & 0 \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & 0 \end{bmatrix}$$

Here,

- the first row contains the second superdiagonal,
- the second row contains the first superdiagonal,
- the third row contains the main diagonal,
- the fourth row contains the first subdiagonal.

LAPACK: banded storage scheme example.

Consider the following banded matrix A with $KL = 1$ and $KU = 2$:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

In **banded storage**, this matrix would be stored in the array `AB` as follows:

$$AB = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} \\ a_{12} & a_{23} & a_{34} & a_{45} & 0 \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & 0 \end{bmatrix}$$

Here,

- the first row contains the second superdiagonal,
- the second row contains the first superdiagonal,
- the third row contains the main diagonal,
- the fourth row contains the first subdiagonal.

LAPACK: routine for banded storage

The LAPACK routine for solving systems of linear equations with banded storage is `?GBSV` for the simple driver and `?GBSVX` for the expert driver.

- These routines take as input the banded matrix in the compact form,
- They perform LU factorization with partial pivoting to solve the system efficiently,
- They are particularly useful when dealing with large banded matrices, as they reduce memory usage and computational time compared to full storage methods.

```
SUBROUTINE ?GBSV( N, NRHS, KL, KU, AB, LDAB, IPIV, B, LDB, INFO )
  INTEGER           N, NRHS, KL, KU, LDAB, LDB, INFO
  INTEGER           IPIV( * )
  ?                 AB( LDAB, * ), B( LDB, * )
END SUBROUTINE ?GBSV
```

LAPACK: routine for banded storage

The LAPACK routine for solving systems of linear equations with banded storage is `?GBSV` for the simple driver and `?GBSVX` for the expert driver.

- These routines take as input the banded matrix in the compact form,
- They perform LU factorization with partial pivoting to solve the system efficiently,
- They are particularly useful when dealing with large banded matrices, as they reduce memory usage and computational time compared to full storage methods.

On exit, details of the factorization: U is stored as an *upper triangular band matrix* with `KL+KU` superdiagonals in rows `1` to `KL+KU+1`, and *the multipliers* used during the factorization *are stored in rows* `KL+KU+2` to `2*KL+KU+1`.

LAPACK: the tridiagonal case

A special case of banded matrices is the **tridiagonal matrix**, which has non-zero elements only on the main diagonal and the first sub- and super-diagonals.
The LAPACK routine for solving systems of linear equations with tridiagonal matrices is `?GTSV` for the simple driver and `?GTSVX` for the expert driver.

```
SUBROUTINE ?GTSV( N, NRHS, DL, D, DU, B, LDB, INFO )
  INTEGER          N, NRHS, LDB, INFO
  ?                DL( * ), D( * ), DU( * ), B( LDB, * )
END SUBROUTINE ?GTSV
```

- `DL` is the subdiagonal elements of size `N-1`,
- `D` is the main diagonal elements of size `N`,
- `DU` is the superdiagonal elements of size `N-1`.

Lapack: packed storage scheme

In the **packed storage scheme**, only the non-zero elements of symmetric or Hermitian matrices are stored in a one-dimensional array. This is particularly useful for large symmetric or Hermitian matrices, as it reduces memory usage significantly.

For a **symmetric matrix**, only the *upper* or *lower* triangular part needs to be stored, as the other part can be inferred due to symmetry.

- → The packed storage format stores the elements column-wise (or row-wise) in a one-dimensional array,
- → This format is efficient for both storage and computation, especially when combined with LAPACK routines designed to work with packed storage.

LAPACK: routine for packed storage

The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER          UPLO
  INTEGER           N, NRHS, LDB, INFO
  ?                 AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

- `UPLO` indicates whether the upper or lower triangular part of the matrix is stored,
- `AP` is the one-dimensional array containing the packed storage of the matrix.

LAPACK: routine for packed storage

The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER          UPLO
  INTEGER           N, NRHS, LDB, INFO
  ?                 AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

- `UPLO` indicates whether the upper or lower triangular part of the matrix is stored,
- `AP` is the one-dimensional array containing the packed storage of the matrix.

If `UPLO = 'U'`, the **upper triangular** part of the matrix is stored column-wise in `AP` as follows:

$$AP = [a_{11}, a_{12}, a_{22}, a_{13}, a_{23}, a_{33}, \ldots, a_{1N}, a_{2N}, \ldots, a_{NN}]$$

LAPACK: routine for packed storage

The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER          UPLO
  INTEGER           N, NRHS, LDB, INFO
  ?                 AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

- `UPLO` indicates whether the upper or lower triangular part of the matrix is stored,
- `AP` is the one-dimensional array containing the packed storage of the matrix.

If `UPLO = 'L'`, the **lower triangular** part of the matrix is stored column-wise in `AP` as follows:

$$AP = [a_{11}, a_{21}, a_{22}, a_{31}, a_{32}, a_{33}, \ldots, a_{N1}, a_{N2}, \ldots, a_{NN}]$$

LAPACK: routine for packed storage

The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER           UPLO
  INTEGER            N, NRHS, LDB, INFO
  ?                  AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

- `UPLO` indicates whether the upper or lower triangular part of the matrix is stored,
- `AP` is the one-dimensional array containing the packed storage of the matrix.
- `N` is the order of the matrix,
- `NRHS` is the number of right hand sides,
- `B` is the right hand side matrix.
- `LDB` is the leading dimension of the array `B`.
- `INFO` is an integer output variable that indicates success or failure of the routine.

LAPACK: routine for packed storage

The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER          UPLO
  INTEGER           N, NRHS, LDB, INFO
  ?                 AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

On **exit**, the solution matrix X overwrites the right hand side matrix B.

The routine uses the Cholesky factorization to solve the system efficiently while taking advantage of the packed storage format. The factor U or L from the Cholesky factorization $A = U^\top U$ or $A = LL^\top$, in the same storage format as A.

LAPACK: Linear Least Squares Problems

The **linear least squares problem** is:

$$\underset{x}{\text{minimize}} \, \|b - Ax\|_2$$

where A is an m-by-n matrix, b is a given m element vector, and x is the n element solution vector.

In the most usual case $m \geq n$ and $\text{rank}(A) = n$, the solution to the problem is unique, and the problem is referred to as finding a least squares solution to an overdetermined system of linear equations.

LAPACK: Linear Least Squares Problems

The **linear least squares problem** is:

$$\underset{x}{\text{minimize}} \, \|b - Ax\|_2$$

where A is an m-by-n matrix, b is a given m element vector, and x is the n element solution vector.

When $m < n$ and $\text{rank}(A) = m$, there are infinitely many solutions x that exactly satisfy $b - Ax = 0$. In this case, it is useful to find the unique solution x which minimizes $\|x\|_2$, referred to as finding a minimum norm solution to an underdetermined system of linear equations.

LAPACK: Linear Least Squares Problems

The **linear least squares problem** is:

$$\underset{x}{\text{minimize}} \, \|b - Ax\|_2$$

where A is an m-by-n matrix, b is a given m element vector, and x is the n element solution vector.

In the general case when $\text{rank}(A) < \min(m, n)$ (i.e., A may be rank-deficient), we seek the minimum norm least squares solution x which minimizes both $\|x\|_2$ and $\|b - Ax\|_2$.

LAPACK: Driver routines for least squares problems

The driver routine `DGELS` solves the linear least squares problem assuming $\text{rank}(A) = \min(m, n)$, i.e., A has **full rank**.

- ➔ It finds a least squares solution of an overdetermined system when $m > n$,
- ➔ It finds a minimum norm solution of an underdetermined system when $m < n$,
- ➔ It uses QR or LQ factorization of A,
- ➔ It allows A to be replaced by $A^\top$ (or A^H if A is complex).

For rank-deficient matrices, several routines are available:

- `DGELSY` — uses **complete orthogonal factorization**,
- `DGELSS` — uses **singular value decomposition** (SVD),
- `DGELSD` — uses **divide-and-conquer SVD** (faster for large problems).

LAPACK: LLS driver routines summary

Operation	Single precision		Double precision	
	real	complex	real	complex
Solve LLS using QR or LQ	SGELS	CGELS	DGELS	ZGELS
Solve LLS using COF	SGELSY	CGELSY	DGELSY	ZGELSY
Solve LLS using SVD	SGELSS	CGELSS	DGELSS	ZGELSS
Solve LLS using DC-SVD	SGELSD	CGELSD	DGELSD	ZGELSD

- All routines can handle multiple right hand sides stored as columns of B and X.
- Each right hand side is solved independently, i.e., we do not find:

$$\arg\min ||B - AX||_2^2.$$

LLS: The different algorithms

2 LAPACK: Linear Algebra PACKage

- **QR and LQ factorization**: $A = QR$ or $A = LQ$ with Q orthogonal and R (or L) triangular.
  - Suitable for full-rank matrices,
  - Efficient for both overdetermined and underdetermined systems,
  - Utilizes orthogonal transformations to minimize numerical errors,
  - Cost: $O(mn^2)$ for $m \geq n$ and $O(nm^2)$ for $m < n$.
- **Complete Orthogonal Factorization (COF)**: $A = Q \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} P^T$, with Q orthogonal, P a permutation matrix, and R_{11} upper triangular, and R_{12} dense,
  - Handles rank-deficient matrices,
  - Decomposes A into orthogonal and triangular matrices,
  - Provides a stable solution even when A is ill-conditioned,
  - Cost: $O(mn^2)$ for $m \geq n$ and $O(nm^2)$ for $m < n$.

LLS: The different algorithms

2 LAPACK: Linear Algebra PACKage

- **Singular Value Decomposition (SVD)**: $A = U\Sigma V^T$, with U and V orthogonal and Σ diagonal,
  - Robust method for rank-deficient matrices,
  - Decomposes A into singular values and vectors,
  - Minimizes the effect of small singular values on the solution,
  - Cost: $O(mn^2 + n^3)$ for $m \geq n$ and $O(nm^2 + m^3)$ for $m < n$.
- **Divide-and-Conquer SVD**:
  - An efficient variant of SVD for large problems,
  - Divides the matrix into smaller submatrices,
  - Combines results to obtain the final solution,
  - Cost: $O(mn^2)$ for $m \geq n$ and $O(nm^2)$ for $m < n$ (faster than traditional SVD for large-scale problems).

LAPACK: Generalized Linear Least Squares Problems

Driver routines are provided for two types of generalized linear least squares problems.

The **first** is the linear equality-constrained least squares problem (LSE):

$$\min_x \|c - Ax\|_2 \quad \text{subject to} \quad Bx = d$$

where $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{p \times n}$, c is a given m-vector, d is a given p-vector, with $p \leq n \leq m + p$.

- The routine `DGGLSE` solves this problem using the **generalized RQ factorization**,
- Assumes B has full row rank p and $\begin{bmatrix} A \\ B \end{bmatrix}$ has full column rank n,
- Under these assumptions, the problem has a unique solution.

LAPACK: Generalized Linear Model Problem

The **second** is the general linear model problem (GLM):

$$\min_x \|y\|_2 \quad \text{subject to} \quad d = Ax + By$$

where $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{n \times p}$, and d is a given n-vector, with $m \leq n \leq m + p$.

- When $B = I$, the problem reduces to an ordinary linear least squares problem,
- When B is square and nonsingular, it is equivalent to the **weighted linear least squares problem**:

$$\min_x \|B^{-1}(d - Ax)\|_2$$

The routine `DGGGLM` solves this problem using the **generalized QR factorization**:

- Assumes A has full column rank m and (A, B) has full row rank n,
- Under these assumptions, there are unique solutions x and y.

LAPACK: Generalized Linear Model Problem

Operation	Single precision		Double precision	
	real	complex	real	complex
Solve LSE using GRQ	SGGLSE	CGGLSE	DGGLSE	ZGGLSE
Solve GLM using GQR	SGGGLM	CGGGLM	DGGGLM	ZGGGLM

- The GRQ decomposes the pair (A, B) into orthogonal and triangular matrices as

$$A = RQ \text{ and } B = ZTQ, \;\; Q, Z \text{ orthogonal}, \;\; R, T \text{ upper triangular}$$

- The GQR decomposes the pair (A, B) into orthogonal and triangular matrices as

$$A = QR \text{ and } B = QTZ, \;\; Q, Z \text{ orthogonal}, \;\; R, T \text{ upper triangular};$$

Eigenproblems and Singular Value Decomposition

2 LAPACK: Linear Algebra PACKage

LAPACK provides a comprehensive set of routines for solving various types of eigenvalue problems and performing singular value decomposition (SVD).

These routines are designed to handle **different matrix types**, including general, symmetric, Hermitian, and banded matrices.

Key features of LAPACK's eigenproblem and SVD routines include:

- Efficient algorithms for computing eigenvalues and eigenvectors,
- Support for both real and complex matrices,
- Routines for computing the SVD of general matrices,
- Specialized routines for symmetric and Hermitian matrices to exploit their properties for improved performance.

LAPACK: Symmetric Eigenproblems (SEP)

The **symmetric eigenvalue problem** is to find the **eigenvalues** λ and corresponding **eigenvectors** $z \neq 0$, such that:

$$Az = \lambda z, \quad A = A^\top, \quad \text{where } A \text{ is real.}$$

For the **Hermitian eigenvalue problem** we have:

$$Az = \lambda z, \quad A = A^H.$$

For both problems the eigenvalues λ are real.

When all eigenvalues and eigenvectors have been computed, we write:

$$A = Z\Lambda Z^\top$$

where Λ is a diagonal matrix whose diagonal elements are the eigenvalues, and Z is an orthogonal (or unitary) matrix whose columns are the eigenvectors. This is the classical **spectral factorization** of A.

Driver routines for symmetric eigenproblems

There are four types of driver routines for symmetric and Hermitian eigenproblems:

- A simple driver (name ending `-EV`) computes all the eigenvalues and (optionally) eigenvectors,
- An expert driver (name ending `-EVX`) computes all or a selected subset of the eigenvalues and eigenvectors,
- A divide-and-conquer driver (name ending `-EVD`) solves the same problem as the simple driver but is much faster for large matrices,
- A **relatively robust representation** (RRR) driver (name ending `-EVR`) is the fastest algorithm and uses the least workspace.

Different driver routines are provided to take advantage of special structure or storage of the matrix A.

LAPACK: Driver routines for symmetric eigenproblems

Function and storage scheme	Single precision		Double precision	
	real	complex	real	complex
simple driver	SSYEV	CHEEV	DSYEV	ZHEEV
divide and conquer driver	SSYEVD	CHEEVD	DSYEVD	ZHEEVD
expert driver	SSYEVX	CHEEVX	DSYEVX	ZHEEVX
RRR driver	SSYEVR	CHEEVR	DSYEVR	ZHEEVR
simple driver (packed)	SSPEV	CHPEV	DSPEV	ZHPEV
divide and conquer (packed)	SSPEVD	CHPEVD	DSPEVD	ZHPEVD
expert driver (packed)	SSPEVX	CHPEVX	DSPEVX	ZHPEVX

LAPACK: Driver routines for symmetric eigenproblems

Function and storage scheme	Single precision		Double precision	
	real	complex	real	complex
simple driver (band)	SSBEV	CHBEV	DSBEV	ZHBEV
divide and conquer (band)	SSBEVD	CHBEVD	DSBEVD	ZHBEVD
expert driver (band)	SSBEVX	CHBEVX	DSBEVX	ZHBEVX
simple driver (tridiagonal)	SSTEV		DSTEV	
divide and conquer (tridiagonal)	SSTEVD		DSTEVD	
expert driver (tridiagonal)	SSTEVX		DSTEVX	
RRR driver (tridiagonal)	SSTEVR		DSTEVR	

Nonsymmetric Eigenproblems (NEP)

The **nonsymmetric eigenvalue problem** is to find the **eigenvalues** λ and corresponding **right eigenvectors** $v \neq 0$, such that:

$$Av = \lambda v.$$

A real matrix A may have complex eigenvalues, occurring as complex conjugate pairs. A vector $u \neq 0$ satisfying:

$$u^H A = \lambda u^H$$

is called a **left eigenvector** of A.

This problem can be solved via the **Schur factorization** of A:

$$A = ZTZ^\top \quad \text{(real case)}, \quad A = ZTZ^H \quad \text{(complex case)}$$

where Z is orthogonal (or unitary) and T is an upper quasi-triangular (or triangular) matrix.

LAPACK: Driver routines for nonsymmetric eigenproblems

Function	Single precision		Double precision	
	real	complex	real	complex
simple driver for Schur factorization	SGEES	CGEES	DGEES	ZGEES
expert driver for Schur factorization	SGEESX	CGEESX	DGEESX	ZGEESX
simple driver for eigenvalues/vectors	SGEEV	CGEEV	DGEEV	ZGEEV
expert driver for eigenvalues/vectors	SGEEVX	CGEEVX	DGEEVX	ZGEEVX

LAPACK: Driver routines for nonsymmetric eigenproblems

Two pairs of drivers are provided:

- `xGEES` and `xGEESX`: compute the Schur factorization of A, with optional ordering of eigenvalues,
- `xGEEV` and `xGEEVX`: compute all eigenvalues and (optionally) right or left eigenvectors.

- The **expert drivers** (`xGEESX` and `xGEEVX`) can additionally balance the matrix and compute condition numbers for the eigenvalues or eigenvectors.

Singular Value Decomposition (SVD)

The **singular value decomposition** of an m-by-n matrix A is given by:

$$A = U\Sigma V^\top \quad (A = U\Sigma V^H \text{ in the complex case})$$

where U and V are orthogonal (unitary) and Σ is an m-by-n diagonal matrix with real diagonal elements σ_i such that:

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m,n)} \geq 0.$$

The σ_i are the **singular values** of A and the first $\min(m, n)$ columns of U and V are the **left** and **right singular vectors** of A.

Driver routines for SVD

Two types of driver routines are provided for the SVD:

- A simple driver `xGESVD` computes all the singular values and (optionally) left and/or right singular vectors,
- A divide-and-conquer driver `xGESDD` solves the same problem but is much faster for large matrices.

- The divide-and-conquer driver uses more workspace but is significantly faster than the simple driver for large matrices.

Generalized Symmetric Definite Eigenproblems (GSEP)

Drivers are provided to compute all the eigenvalues and (optionally) the eigenvectors of the following types of problems:

1. $Az = \lambda Bz$
2. $ABz = \lambda z$
3. $BAz = \lambda z$

where A and B are symmetric or Hermitian and B is positive definite.

For all these problems the eigenvalues λ are real. The matrices Z of computed eigenvectors satisfy:

- $Z^\top AZ = \Lambda$ (problem types 1 and 3) or $Z^{-1}AZ^{-\top} = I$ (problem type 2),
- $Z^\top BZ = I$ (problem types 1 and 2) or $Z^\top B^{-1}Z = I$ (problem type 3),

where Λ is a diagonal matrix with the eigenvalues on the diagonal.

Driver routines for GSEP

Three types of driver routines are provided for generalized symmetric and Hermitian eigenproblems:

- A simple driver (name ending `-GV`) computes all the eigenvalues and (optionally) eigenvectors,
- An expert driver (name ending `-GVX`) computes all or a selected subset of the eigenvalues and eigenvectors. If few enough eigenvalues or eigenvectors are desired, the expert driver is faster than the simple driver,
- A divide-and-conquer driver (name ending `-GVD`) solves the same problem as the simple driver but is much faster for large matrices, although it uses more workspace.

Different driver routines are provided to take advantage of special structure or storage of the matrices A and B.

LAPACK: Driver routines for GSEP

Function and storage scheme	Single precision		Double precision	
	real	complex	real	complex
simple driver	SSYGV	CHEGV	DSYGV	ZHEGV
divide and conquer driver	SSYGVD	CHEGVD	DSYGVD	ZHEGVD
expert driver	SSYGVX	CHEGVX	DSYGVX	ZHEGVX
simple driver (packed)	SSPGV	CHPGV	DSPGV	ZHPGV
divide and conquer (packed)	SSPGVD	CHPGVD	DSPGVD	ZHPGVD
expert driver (packed)	SSPGVX	CHPGVX	DSPGVX	ZHPGVX
simple driver (band)	SSBGV	CHBGV	DSBGV	ZHBGV
divide and conquer (band)	SSBGVD	CHBGVD	DSBGVD	ZHBGVD
expert driver (band)	SSBGVX	CHBGVX	DSBGVX	ZHBGVX

Generalized Nonsymmetric Eigenproblems (GNEP)

Given a matrix pair (A, B), where A and B are square $n \times n$ matrices, the **generalized nonsymmetric eigenvalue problem** is to find the **eigenvalues** λ and corresponding **eigenvectors** $x \neq 0$ such that:

$$Ax = \lambda Bx$$

or to find the eigenvalues μ and corresponding eigenvectors $y \neq 0$ such that:

$$\mu Ay = By$$

These problems are equivalent with $\mu = 1/\lambda$ and $x = y$ if neither λ nor μ is zero.

Matrix Pencils and Eigenvalue Representation

2 LAPACK: Linear Algebra PACKage

To deal with cases where λ or μ is zero or nearly so, LAPACK routines return two values, α and β, for each eigenvalue, such that:

$$\lambda = \alpha/\beta \quad \text{and} \quad \mu = \beta/\alpha$$

Vectors $u \neq 0$ or $v \neq 0$ satisfying:

$$u^\top A = \lambda u^\top B \quad \text{or} \quad \mu v^\top A = v^\top B$$

are called **left eigenvectors**.

The matrix pencil $A - \lambda B$ is used to refer to the generalized eigenproblem. The problem is called:

- **Regular** if $\det(A - \lambda B) \not\equiv 0$ for all λ,
- **Singular** if $\det(A - \lambda B) \equiv 0$ for all λ (signaled by $\alpha = \beta = 0$).

Generalized Schur Decomposition

The generalized nonsymmetric eigenvalue problem can be solved via the **generalized Schur decomposition** of the matrix pair (A, B).

In the **real case**:

$$A = QSZ^\top, \quad B = QTZ^\top$$

In the **complex case**:

$$A = QSZ^H, \quad B = QTZ^H$$

where Q and Z are orthogonal (or unitary), T is upper triangular, and S is upper quasi-triangular with 1×1 and 2×2 diagonal blocks.

The columns of Q and Z are called **left and right generalized Schur vectors** and span pairs of **deflating subspaces** of A and B.

Driver routines for GNEP

Two pairs of drivers are provided:

- `xGGES` and `xGGESX`: compute the generalized Schur decomposition of (A, B), with optional ordering of eigenvalues,
- `xGGEV` and `xGGEVX`: compute all generalized eigenvalues and (optionally) right or left eigenvectors.

The **expert drivers** (`xGGESX` and `xGGEVX`) can additionally:

- Balance the matrix pair to improve conditioning,
- Compute condition numbers for the eigenvalues or eigenvectors.

Summary of GNEP driver routines

Function	Single precision		Double precision	
	real	complex	real	complex
simple driver (Schur)	SGGES	CGGES	DGGES	ZGGES
expert driver (Schur)	SGGESX	CGGESX	DGGESX	ZGGESX
simple driver (eigenvalues)	SGGEV	CGGEV	DGGEV	ZGGEV
expert driver (eigenvalues)	SGGEVX	CGGEVX	DGGEVX	ZGGEVX

Generalized Singular Value Decomposition (GSVD)

The **generalized (or quotient) singular value decomposition** of an $m \times n$ matrix A and a $p \times n$ matrix B is given by:

$$A = U\Sigma_1[0, R]Q^\top \quad \text{and} \quad B = V\Sigma_2[0, R]Q^\top$$

where:

- U is $m \times m$, V is $p \times p$, Q is $n \times n$, all orthogonal (or unitary for complex),
- R is $r \times r$, upper triangular and nonsingular, where r is the rank of $\begin{bmatrix} A \\ B \end{bmatrix}$,
- Σ_1 is $m \times r$ and Σ_2 is $p \times r$, both real, nonnegative, and diagonal.

The ratios $\alpha_1/\beta_1, \ldots, \alpha_r/\beta_r$ are called the **generalized singular values** of the pair (A, B).

Special Cases of GSVD

Important special cases of the generalized singular value decomposition include:

- If B is square and nonsingular, then $r = n$ and the GSVD is equivalent to the SVD of AB^{-1},
- If the columns of $\begin{bmatrix} A^\top & B^\top \end{bmatrix}^\top$ are orthonormal, then $r = n$, $R = I$, and the GSVD is equivalent to the **Cosine-Sine (CS) decomposition**,
- The generalized eigenvalues of $A^\top A - \lambda B^\top B$ can be expressed in terms of the GSVD.

Driver routine for GSVD

A single driver routine `xGGSVD` computes the generalized singular value decomposition of A and B.

Function	Single precision		Double precision	
	real	complex	real	complex
GSVD	SGGSVD	CGGSVD	DGGSVD	ZGGSVD

- The method is based on the generalized QR and RQ factorizations,
- It is useful for solving certain least squares and generalized eigenvalue problems,
- The GSVD provides a unified framework for understanding relationships between different matrix decompositions.

Computational Routines

2 LAPACK: Linear Algebra PACKage

Driver routines call a sequence of **computational routines** (also called auxiliary routines) to perform specific tasks:

Factorization routines compute matrix factorizations (LU, QR, Cholesky, etc.),

Solver routines solve systems using a precomputed factorization,

Eigenvalue routines reduce matrices to condensed forms (tridiagonal, Hessenberg),

Utility routines perform auxiliary operations (scaling, orthogonal transformations).

- Users can call computational routines directly for **finer control** and **better performance**,
- Useful when you need to **reuse a factorization** for multiple operations,
- Allows **combining different algorithms** in a custom sequence.

LAPACK: Factorization Routines

Examples of factorization computational routines:

- `xGETRF`: LU factorization with partial pivoting,
- `xGETRS`: solve using LU factorization,
- `xGEQRF`: QR factorization,
- `xGERQF`: RQ factorization,
- `xGELQF`: LQ factorization,
- `xORMQR` (or `xUNMQR`): apply orthogonal transformation from QR,
- `xPOTRF`: Cholesky factorization.

Combining computational routines allows implementing **custom algorithms**:

```
! Compute QR factorization
call dgeqrf(m, n, A, lda, tau, work, lwork, info)
! Apply Q to a vector
call dormqr('L', 'T', m, nrhs, n, A, lda, tau, B, ldb, work, lwork, info)
```

LAPACK: Solver Routines

Solver routines use precomputed factorizations to solve linear systems efficiently:

- `xGETRS`: solve using LU factorization from `xGETRF`,
- `xPOTRS`: solve using Cholesky factorization from `xPOTRF`,
- `xTRTRS`: solve triangular systems,
- `xGBTRS`: solve banded systems using factorization from `xGBTRF`,
- `xGTTRS`: solve tridiagonal systems using factorization from `xGTTRF`.

These routines typically perform $O(n^2)$ operations instead of $O(n^3)$ for factorization:

```
! Factorize once
call dgetrf(n, n, A, lda, ipiv, info)
! Solve for multiple right-hand sides
do i = 1, nrhs
  call dgetrs('N', n, 1, A, lda, ipiv, B(:,i), n, info)
end do
```

LAPACK: Solver Routines

Solver routines use precomputed factorizations to solve linear systems efficiently:

- `xGETRS`: solve using LU factorization from `xGETRF`,
- `xPOTRS`: solve using Cholesky factorization from `xPOTRF`,
- `xTRTRS`: solve triangular systems,
- `xGBTRS`: solve banded systems using factorization from `xGBTRF`,
- `xGTTRS`: solve tridiagonal systems using factorization from `xGTTRF`.

- Separating factorization and solving provides significant performance gains for multiple systems,
- Essential for iterative refinement and other advanced techniques.

LAPACK: Eigenvalue Reduction and Schur Manipulation

Reducing to Hessenberg form is often the first step in computing eigenvalues of nonsymmetric matrices:

- `xGEHRD`: reduce a general matrix to upper Hessenberg form,
- `xORGHR` (or `xUNGHR`): generate the orthogonal matrix from the reduction,
- Hessenberg form has zeros below the first subdiagonal, reducing subsequent eigenvalue computations.

Schur factorization manipulation:

- `xTREXC`: reorder eigenvalues in Schur factorization by exchanging diagonal blocks,
- `xTRSEN`: reorder and compute condition numbers for selected eigenvalues,
- Useful for isolating specific eigenvalues or organizing them by magnitude or stability.

LAPACK: Eigenvalue Reduction and Schur Manipulation

Solving Sylvester equations:

- `xTRSYL`: solve the generalized Sylvester equation $AX + XB = C$ or related forms,
- Requires Schur factorizations of A and B as input,
- Applications include model reduction, control theory, and matrix equation solutions.

```
! Reduce to Hessenberg form
call dgehrd(n, 1, n, A, lda, tau, work, lwork, info)
! Reorder Schur form
call dtrexc('V', n, T, ldt, Q, ldq, ifst, ilst, work, info)
! Solve Sylvester equation: AX + XB = C
call dtrsyl('N', 'N', 1, m, n, A, lda, B, ldb, C, ldc, scale, info)
```

Why Use Computational Routines?

Factorization reuse Compute factorization once, use it for multiple right-hand sides or operations

Memory efficiency Control allocation and reuse of intermediate arrays

Algorithm customization Combine different routines to implement specialized algorithms

Performance Avoid redundant computations when multiple related operations are needed

Stability control Apply equilibration or scaling before factorization for better conditioning

Example use case: Solve multiple systems $Ax_i = b_i$ where b_i depends on x_{i-1}:

1. Compute LU factorization of A once using `DGETRF`,
2. For each b_i, call `DGETRS` with the precomputed factors,
3. Much faster than calling `DGESV` repeatedly.

Conclusions

3 Conclusions and next step

- LAPACK provides a suite of routines for solving LA problems efficiently,
- Driver routines offer high-level interfaces for common tasks,
- Computational routines allow fine-grained control,
- Understanding both types of routines enables users to optimize performance and tailor solutions to specific needs.

- Next step: look at the **ScaLAPACK** library for distributed-memory parallel computing,
- ScaLAPACK extends LAPACK's capabilities to large-scale problems across multiple processors.